**Ministry of Education, Culture, and Research of the Republic of Moldova**

**Technical University of Moldova**

**The Faculty of Computers, Informatics, and Microelectronics**

**REPORT**

Laboratory work no.0

*Topic: SOLID Principles*

Author:

st. gr. FAF-223 Gliga Daniela

Chişinău - 2024

**Objectives:**

* Study and understand SOLID principles
* Learn how to implement them

**Theory:**

The SOLID principles are 5 essential guidelines that help in improving software design, making code more maintainable and scalable. These 5 principles are: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion.

**Single Responsability Principle** states that a class should have only one reason to change, which means that each class should have basically one job.

**Open/Closed Principle** states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. In other words a class should be able to extend its behaviour without modifying the the code.

**Liskov Principle** states that derived or child classes must be substitutable for their base or parent classes. This means that, given that class B is a subclass of class A, we should be able to pass an object of class B to any method that expects an object of class A and the method should not give any weird output in that case.

**Interface Segregation Principle** states that do not force any client to implement an interface which is irrelevant to them.It is basically about separating interfaces.

**Dependency Inversion Principle** states that our classes should depend upon interfaces or abstract classes instead of concrete classes and functions.

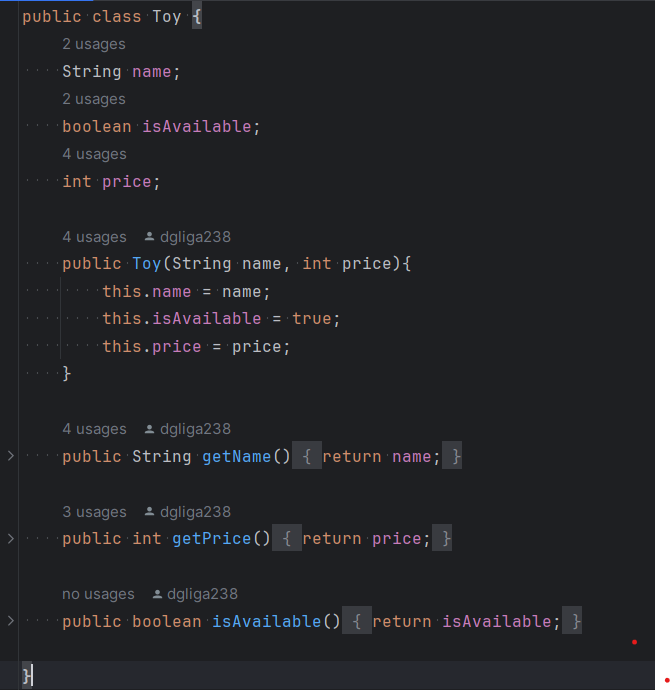
**Task:**

Implement 2 SOLID letters in a simple project.

**Implementation:**

The following project simulates the work in a toy shop (just simple basic logic), implementing Single Responsibility Principle and Interface Segregation Principle. In the following lines I will explain how each of 2 principles is implemented and used specifically for this project.

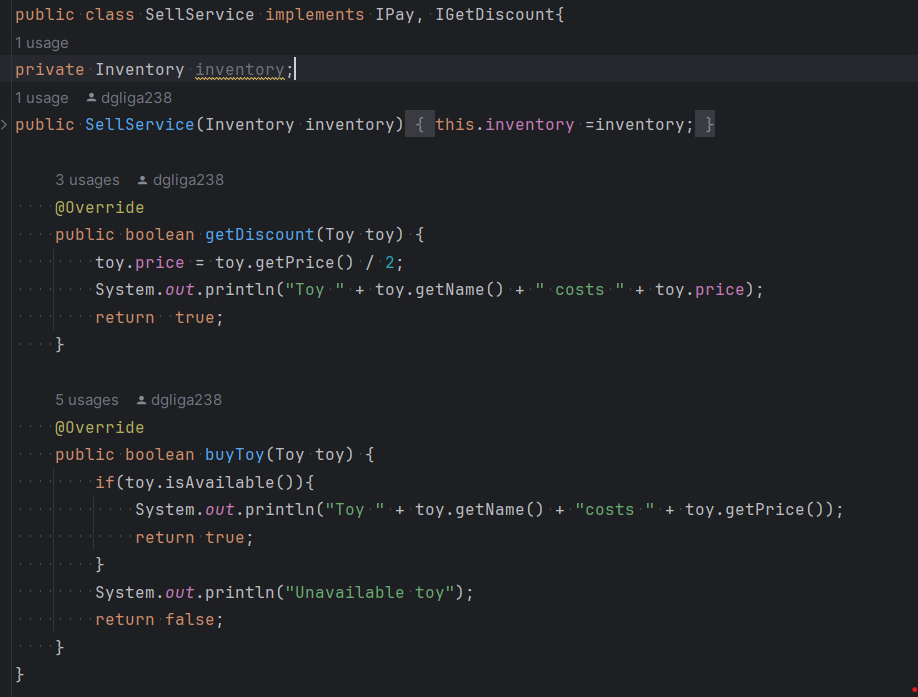
From the definition Single Responsibility Principle , every class should have just one job, therefore I created one class for defining the Toy itself, including specific methods which take the information about each toy (name, availability and price):



Two more classes created for separating the responsabilities inside the shop are: Inventoy class, which deals with managing the inventory which adds / removes toys and displays all the toys. It does not deal with selling them or calculating fees.

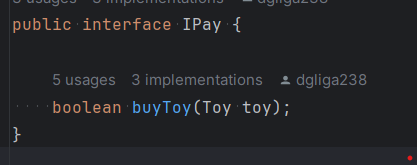


and the SellService class which is responsible for selling toys and applying discounts. Two methods implemented here: buyToy() implements the purchase operation and the getDiscount() method implements reducing the cost for some specific clients.:

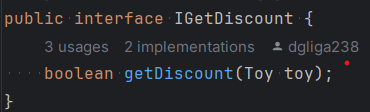


Also, for separating the roles, two different classes were created for regular customers and for Permanent customers since they have some features which are different. For this purpose also was implemented further the Interface Segregation Principle.

The Interface Segregation Principle suggests that clients should not be forced to depend on interfaces they do not use. In this project two separate interfaces were created: IPay and IGetDiscount.



This interface (IPay) defines the buyToy method, which is essential for any class that represents an entity capable of purchasing a toy. Both RegularCustomer and PermanentCustomer implement this interface because they need the ability to buy toys.



This interface (IGetDiscount) defines the getDiscount method, which is relevant for entities that can receive a discount. Only PermanentCustomer implements this interface, as regular customers are not supposed to have access to discount functionalities.

**Conclusion:**

In conclusion, by following the Interface Segregation Principle (ISP) and Single Responsibility Principle (SRP), this project shows a well-structured design. The separation of responsibilities into separate classes like Toy, Inventory, SellService, and customer classes makes the system easier to maintain and modify. Additionally, using interfaces such as IPay and IGetDiscount ensures that each class only implements what it needs, which means that everyone gets straightforward their features, without messing with others. This approach helps in making the code more modular, flexible, and easier to extend with new features, improving the overall maintainability and scalability of the system.